

# UC Irvine

## ICS Technical Reports

### Title

Analyzing safety and fault tolerance using time Petri nets

### Permalink

<https://escholarship.org/uc/item/230423kh>

### Authors

Leveson, Nancy G.

Stolzy, Janice T.

### Publication Date

1984

Peer reviewed

Notice: This Material  
may be protected  
by Copyright Law  
(Title 17 U.S.C.)

Z  
699  
C3  
no. 220

**Analyzing Safety and Fault Tolerance Using Time Petri  
Nets**

*Nancy G. Leveson*  
*Janice L. Stolz*

Technical Report #220

Department of Information and Computer Science  
University of California, Irvine  
Irvine, California 92717

July 1984

This work was supported in part by a joint grant of the  
University of California MICRO Committee and Hughes Aircraft  
Co.

### ABSTRACT

The application of time Petri net modelling and analysis techniques to safety-critical real-time systems is explored and procedures described which allow analysis of safety, recoverability, and fault tolerance. These procedures can be used to help determine software requirements, to guide the use of fault detection and recovery procedures, to determine conditions which require immediate mitigating action to prevent accidents, etc. Thus it is possible to establish important properties during the synthesis of the system and software design instead of using guesswork and costly *a posteriori* analysis.

# Analyzing Safety and Fault Tolerance Using Time Petri Nets<sup>†</sup>

Nancy G. Leveson  
Janice L. Stolzy

Information and Computer Science  
University of California, Irvine  
Irvine, California 92717  
(714) 856-7403

## Introduction

Computers are increasingly being used as passive (monitoring) and active (controlling) components of real-time systems, e.g. air traffic control, aerospace, aircraft, industrial plants, and hospital patient monitoring systems. The problems of safety become important when these applications include systems where the consequences of failure are serious and may involve grave danger to human life and property.

Although in a batch system it is reasonable to abort execution and attempt to fix the problem when a failure occurs, control usually cannot be abandoned abruptly in an embedded system. Therefore, responses to hardware failures, software faults, human error, and undesired and perhaps unexpected environmental conditions must be built into the system. These responses can take three basic forms:

- 1) a *fault-tolerant* system continues to provide full performance and functional capabilities in the presence of operational faults.
- 2) a *fail-soft* system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed.
- 3) a *fail-safe* system attempts to limit the amount of damage caused by a failure. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

These responses are, for most situations, in the order of decreasing desirability although when the functional and safety requirements of the system are not identical (and especially when they are conflicting), they are not necessarily of

---

<sup>†</sup>This work was partially supported by a MICRO grant co-funded by the University of California and Hughes Aircraft Co.

decreasing importance.

The area of system safety is well-established, and procedures exist to identify and analyze electromechanical hazards along with techniques to eliminate or limit hazards in the final product (for a summary, see Hammer (1972)). Unfortunately, much more is known about how to engineer safe mechanical systems than safe software systems. With the increased use of software in safety-critical components of complex systems, government certification agencies and contractors are increasingly including requirements for software hazard analysis and verification of software safety (e.g. see MIL-STD-882b: System Safety Program Requirements). Modelling and analysis tools are desperately needed to aid in these tasks. This paper explores the application of Petri net modelling and analysis techniques to the design of safety-critical real-time systems. Because timing is crucial with respect to the control of real-time systems, Time Petri nets are used.

The next section describes the general approach to be taken. Following that, procedures are outlined for eliminating hazards from the system design. Then potential failures are added to the analysis procedures.

## Safety Analysis

Whereas *system reliability* deals with the problems of ensuring that a system, including all hardware and software subsystems, performs a required task or mission for a specified time in a specified environment, *system safety* is concerned only with ensuring that a mishap does not occur in the process. Usually there are many possible system failures which have relatively little "cost" associated with them. Others have such drastic consequences that an attempt must be made to avoid them at all costs, perhaps even at the cost of attaining some or all of the goals of the system.<sup>†</sup> For example, an amusement park ride may have to be temporarily stopped because conditions are such (e.g. a foreign object is on the tracks) that a derailment is possible. Thus the response to a safety critical failure may focus on reduction of risk rather than attainment of mission [Leveson (1984)].

While software itself cannot be unsafe, it can issue commands to a system it controls which place the system in an unsafe state. Furthermore, the controlling software should be able to detect when factors beyond the control of the

---

<sup>†</sup>In a system whose sole purpose is the sustaining of life, e.g. a pacemaker, these conflicts between safety and other system requirements do not occur.

computer (e.g. environmental conditions) place the system in a hazardous state and to take steps to eliminate the hazard or, if that is not possible, initiate procedures to minimize the hazard. This then is the problem of *software safety*.

If software safety is to be studied and used as a measure of software quality, then some definitions are necessary. A *mishap* is an event or series of events which results in death, injury, illness, or damage to or loss of property or equipment. A *hazard* or *unsafe state* is a condition or state of the system with the potential for (i.e. some non-zero probability of) leading to a mishap. Hazards can be categorized by the aggregate probability of the occurrence of the individual conditions which make up the hazard and by the seriousness of the resulting mishap. Together these constitute *risk*.

The first step in an safety analysis is to identify the system hazards and assess their severity and probability (i.e. risk). The next step is to design the system so as to eliminate hazards or (if that is not possible) to minimize the risk by altering the design so that there is very little probability of the hazard occurring. This can be accomplished by first ensuring that the system as specified is safe, i.e. given that the specifications are correctly implemented and no failures occur, operation of the system will not result in a mishap. The next step in the design process is to identify and eliminate (by using fault tolerance techniques) single point failure modes which can lead to a hazard. Finally, techniques are used to ensure that the probability of multiple sequences of failures leading to a hazard is sufficiently low. If it is impossible to completely eliminate the possibility of a hazard, a design goal may be to minimize the effects of the hazard should it occur. In this case the system should detect the hazard and attempt to eliminate it, if possible; otherwise an attempt should be made to minimize any possible effects. In either case, in order to reduce risk, the *exposure time* (length of time of occurrence) of the hazardous conditions must be minimized. The goal of the techniques presented in this paper is to develop formal procedures to aid in this safety analysis process.

It is important to stress the "system" nature of the problem. Software does not harm anyone -- only the instruments which it controls can do damage. Therefore, software safety procedures cannot be developed in a vacuum, but must be considered as part of the overall system safety. For example, a particular software error may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, an environmental event or failure may be involved in the software error. Mishaps are often the result of multiple failure sequences which involve hardware, software, and human failures. One modelling technique which has the potential for analyzing software for real-time systems

within a system viewpoint is Time Petri nets.

Petri nets have been used to model and analyze systems for such properties as deadlock and reachability. In this paper we show how they can be used in designing and analyzing such properties as safety and fault-tolerance. Time Petri nets were chosen over other modeling techniques for the following reasons:

- They can be used early in the development cycle (vs. verification and validation methods) so that changes can be made while they are still relatively inexpensive. In practice, very safe systems, once built and tested, are sometimes found to be unacceptable in terms of performance and/or reliability. The design must then be modified. It would be much less costly to determine these tradeoffs using a modeling technique before the system is actually built and to adjust the design accordingly. Performance analysis using time Petri nets has been explored by Ramamoorthy and Ho (1980), Razouk (1983), and Zuberek (1980).
- A systems approach is possible with Petri nets since hardware [e.g. Azema *et.al* (1976), Hack (1972)], software [e.g. Nelson *et.al* (1983), Peterson (1981), Rose (1984)], and human behavior can be modelled using the same language. By combining hardware, software, and human components within one model, it is possible to determine, for example, the effects of a failure or fault in one component on another component. It is also possible to use the model to determine software safety and fault tolerance requirements. Writing correct software requirements is a difficult problem for which there are few analytical tools available. The possibility of using Petri nets for this problem is intriguing. Techniques such as Failure Modes and Effects Analysis (FMEA) and Preliminary Hazard Analysis (PHA) [Hammer (1972)] have been developed to determine the system safety requirements. However, there is a need to be able to go from the system safety requirements to the software safety requirements. Fault tree analysis has been used for the task [Leveson and Harvey (1983)], but has some drawbacks. Using the hazardous states which have been identified in the PHA, it may be possible to work backward to the software interface using Petri net analysis techniques and thus to derive the software safety requirements. As an example, Coolahan and Roussopoulos (1983) have shown how Petri nets can be used to derive timing requirements for modules in real-time systems where the service involves repetitive performance of similar activities at a fixed, constant, and predetermined interval. A more general procedure which does not have these limitations can be found in Merlin (1974) and Merlin and Farber (1976). Petri net models can also be used to determine the most critical software functions

which can then be augmented with fault tolerance facilities and to determine the conditions which must be incorporated into the run-time tests associated with these facilities such as watchdog timers and acceptance tests in recovery blocks.

- Petri nets can be used at various levels of abstraction [e.g. Azema *et.al* (1975), Estrin (1978), Noe (1978)]. Thus an event at one level of abstraction may be a series of events at a lower level of abstraction. The ability to analyze the effects of failures at one level on higher levels of the system would be useful.
- Petri nets provide a modelling language which can be used for both formal analysis and simulation [Razouk (1981)]. Questions about the design which cannot be answered through analysis techniques can often be answered through simulation. Interactive simulators also provide the designer with the opportunity to study potential design modifications. Finally, the non-software components of the Petri net model can be used as a test harness for software testing and debugging.
- Using Time Petri allows the incorporation of timing information into the analysis. Many modelling and analysis techniques do not have this facility, but it is imperative for real-time embedded systems. In these systems, for example, basically correct software actions which are too early or too late can lead to unsafe conditions. Any safety analysis which does not include time will be incomplete.

It is important to note that Petri net models as used in this paper are primarily concerned with modeling control and control failures. Obviously, there are important errors and failures which involve other than the control aspects of a system, e.g. data and computation problems. These can be modeled by augmenting the basic Petri net model with a data graph and interpretation as has been done in Sara [Estrin (1978)] and Logos [Rose and Albarran (1975)]. This is an important extension but is beyond the scope of this paper.

Now that the basic goals of software safety analysis have been described and the reasons for choosing this particular technique outlined, the details of the procedures can be presented.



## Formal Definitions

A Petri net is composed of a set of *places*  $P$ , a set of *transitions*  $T$ , an *input function*  $I$ , an *output function*  $O$ , and an *initial marking*  $\mu_0$ . The input function  $I$  is a mapping from the transition  $t_i$  to a bag of places  $I(t_i)$  where a bag is a generalization of a set which allows multiple occurrences of an element. Similarly, the output function  $O$  maps a transition  $t_i$  to a bag of places  $O(t_i)$ . The initial placement of tokens on the places of the net is specified by  $\mu_0$ . Formally, this is written:

**Definition:** A *Petri net structure*,  $\Phi$ , is a five-tuple,  $\Phi=(P,T,I,O,\mu_0)$ .

$P=\{p_1, p_2, \dots, p_n\}$  is a finite set of *places*,  $n \geq 0$ .

$T=\{t_1, t_2, \dots, t_m\}$  is a finite set of *transitions*,  $m \geq 0$ . The set of places and the set of transitions are disjoint,  $P \cap T = \emptyset$ .

$I: T \rightarrow P^\infty$  is the *input function*, a mapping from transitions to bags of places.

$O: T \rightarrow P^\infty$  is the *output function*, a mapping from transitions to bags of places.

Finally,  $\mu_0: P \rightarrow N$  is the *initial marking* for the net where  $N$  is the set of non-negative integers.

**Definition:** The *multiplicity* of an input place  $p_i$  for a transition  $t_j$  is the number of occurrences of the place in the input bag of the transition, denoted  $\#(p_i, I(t_j))$ . The multiplicity of an output place is defined similarly and denoted  $\#(p_i, O(t_j))$ .

A graph structure is often used for illustration of Petri nets where a *circle* "O" represents a place and a *bar* "|" represents a transition. Figure 1 shows a petri net and the corresponding petri net graph. An arrow from a place to a transition defines the place to be an input to the transition. Similarly, an output place is indicated by an arrow from the transition to the place.

The dynamic aspects of Petri net models are denoted by markings which are assignments of *tokens* to the places of a Petri net. Markings may change during *execution* of a Petri net.

**Definition:** A *marking*  $\mu$  of a Petri net  $\Phi$  is a function from the set of places  $P$  to the nonnegative integers  $N$ ,  $\mu: P \rightarrow N$ .

The execution of a Petri net is controlled by the number and distribution of tokens in the Petri net.

**Definition:** A transition is enabled if and only if each of its input places contains at least as many tokens as there exists arcs from that place to the transition, i.e.

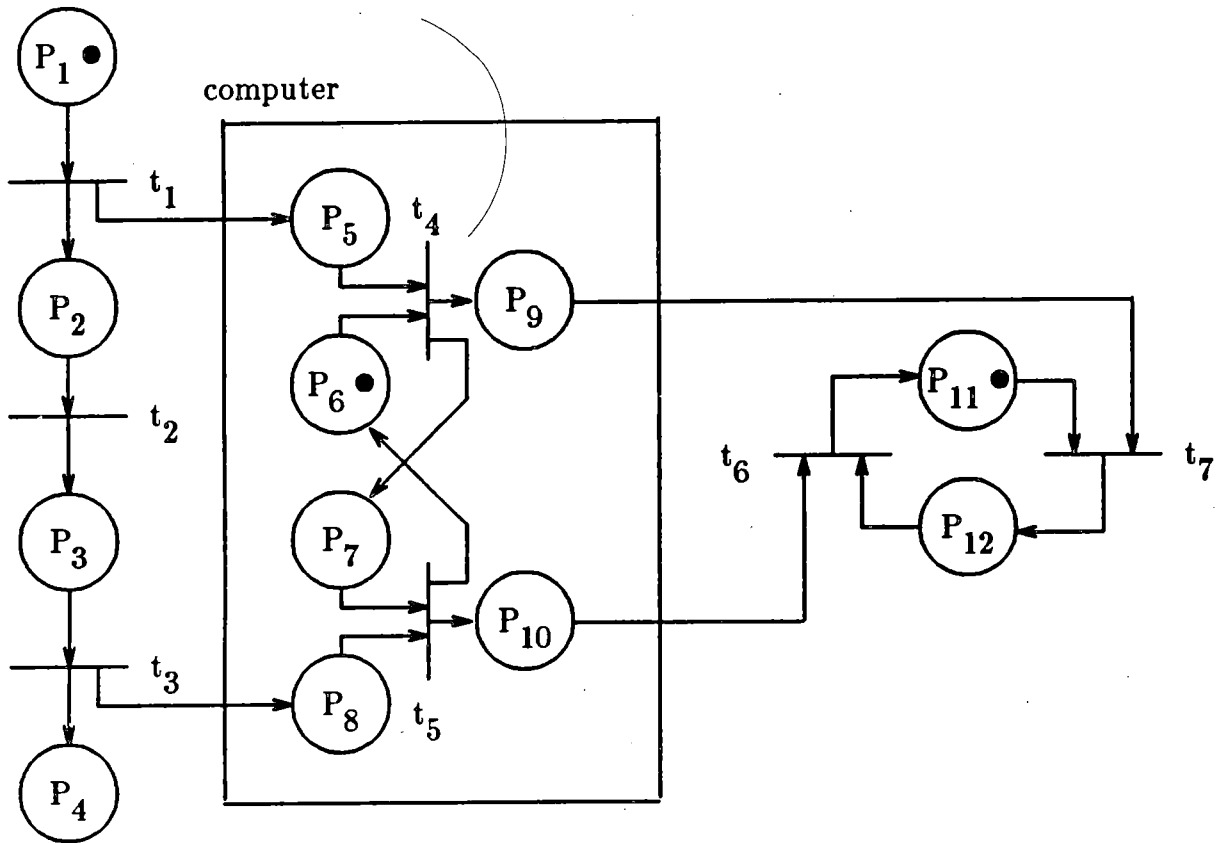


Figure 1. A Petri Net Graph

$$P = \{ P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{10}, P_{11}, P_{12} \}$$

$$T = \{ t_1, t_2, t_3, t_4, t_5, t_6, t_7 \}$$

$$\mu_0 = (1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0)$$

$I(t_1) = \{P_1\}$	$O(t_1) = \{P_2, P_5\}$
$I(t_2) = \{P_2\}$	$O(t_2) = \{P_3\}$
$I(t_3) = \{P_3\}$	$O(t_3) = \{P_4, P_8\}$
$I(t_4) = \{P_5, P_6\}$	$O(t_4) = \{P_7, P_9\}$
$I(t_5) = \{P_7, P_8\}$	$O(t_5) = \{P_6, P_{10}\}$
$I(t_6) = \{P_{10}, P_{12}\}$	$O(t_6) = \{P_{11}\}$
$I(t_7) = \{P_9, P_{11}\}$	$O(t_7) = \{P_{12}\}$

Figure 1b. Description of the Above Petri Net

$$\mu(p_i) \geq \#(p_i, I(t_j)) \text{ for all } p_i \in P$$

When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places. Given the Petri net marking in figure 1, the next state after firing transition  $t_1$  is shown in figure 2. Transition firings continue as long as there exists at least one enabled transition.

When using Petri nets to model systems, places represent conditions and transitions are used to represent events. Figure 1 can be interpreted as a model of a simple railroad crossing.  $P_1, P_2, P_3$ , and  $P_4$  represent the different conditions that can hold for the train (i.e. approaching, just before, within, and past the crossing, respectively). Similarly, transitions 1, 2, and 3 denote the events of signalling the train's approach, entering the crossing, and signalling the train's departure. The large box represents the controlling device or computer -- either hardware or software based. The states of the gate are represented by two places  $P_{11}$  (the gate is up) and  $P_{12}$  (the gate is down). Transitions 6 and 7 represent the events of raising and lowering the gate respectively.

The state of the Petri net (and hence the state of the modeled system) is defined by the marking (the existing conditions). The change in state caused by firing a transition is defined by the *next-state function*  $\delta$ .

**Definition:** The *next-state function*  $\delta: N^n \times T \rightarrow N^n$  for a Petri net  $\Phi=(P,T,I,O,\mu_0)$  with marking  $\mu$  and transition  $t_j \in T$  is defined if and only if  $t_j$  is enabled.

If  $\delta(\mu, t_j)$  is defined, then  $\delta(\mu, t_j) = \mu'$  where

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)) \text{ for all } p_i \in P$$

**Definition:** For a Petri net  $\Phi=(P,T,I,O,\mu_0)$  with marking  $\mu$ , a marking  $\mu'$  is *immediately reachable* from  $\mu$  if there exists a transition  $t_j \in T$  such that  $\delta(\mu, t_j) = \mu'$ .

The "reachability" relationship is the reflexive transitive closure of the "immediately reachable" relationship.

**Definition:** The *reachability set*  $R(\Phi, \mu)$  for a Petri net  $\Phi=(P,T,I,O,\mu_0)$  with marking  $\mu$  is the smallest set of markings defined by:

1.  $\mu \in R(\Phi, \mu)$
2. If  $\mu' \in R(\Phi, \mu)$  and  $\mu'' = \delta(\mu', t_j)$ , for some  $t_j \in T$ , then  $\mu'' \in R(\Phi, \mu)$ .

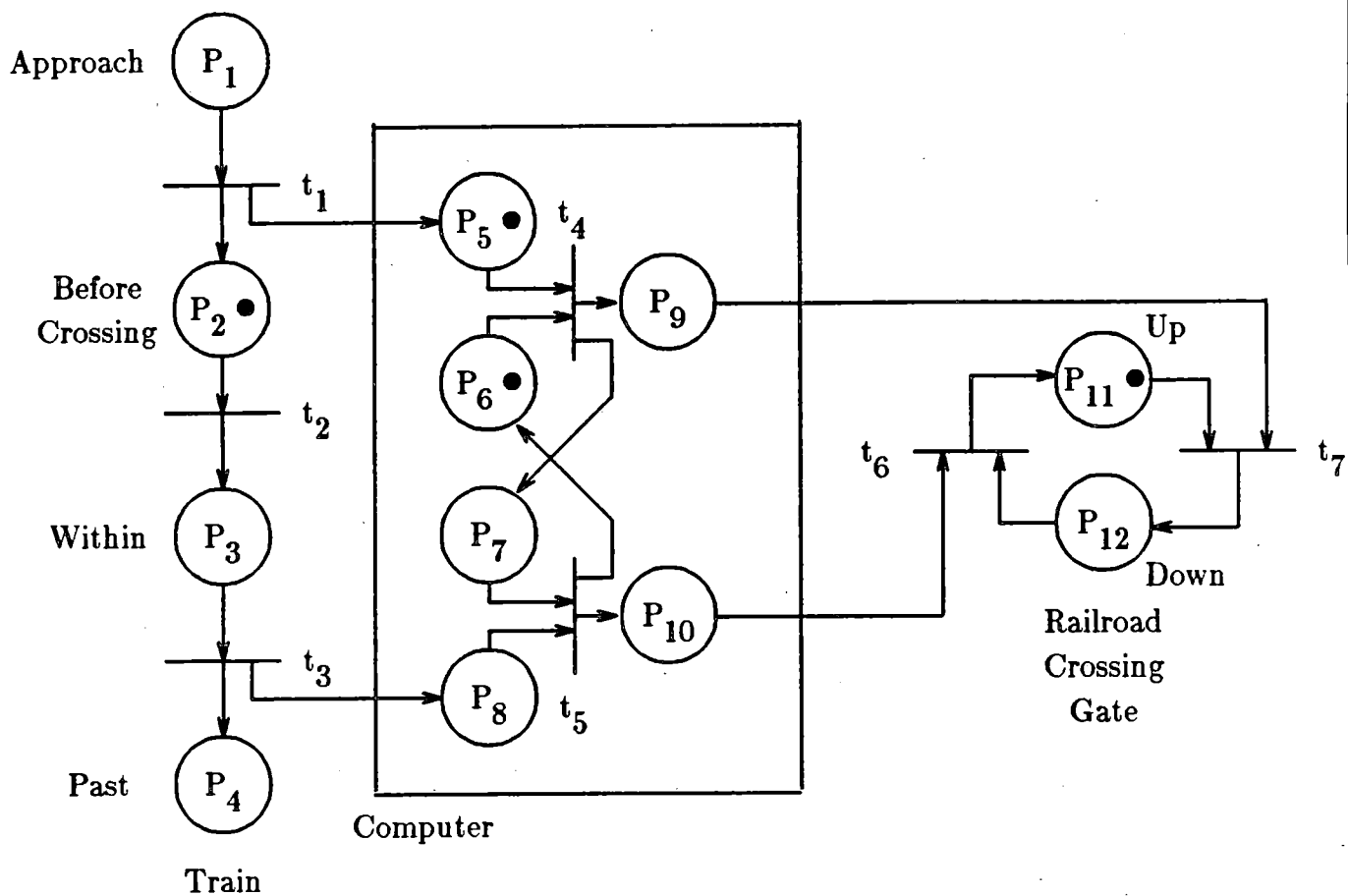


Figure 2. A Petri Net Graph with the Next State Shown

Both trees and graphs have been used to represent the reachability state. In this paper, a reachability graph is used where the nodes of the graph are labeled with the present marking (i.e. the state) and the arcs represent transitions between states (see figure 3).

**Definition:** A path  $P$  in the reachability graph is a sequence of transitions  $t_1, \dots, t_j$  starting at marking  $\mu_{i-1}$  to  $\mu_j$  such that

$$\delta(\mu_{n-1}, t_n) = \mu_n \text{ for } n = i..j$$

**Definition:** The *extended next-state function*  $\delta^*$  is defined for a marking  $\mu$ , and a sequence of transitions  $s \in T^*$  by

$$\delta^*(\mu, t_j.s) = \delta(\delta^*(\mu, t_j), s)$$

$$\delta^*(\mu, \lambda) = \mu$$

To model time requires enhancements to the basic Petri net model. There have been several proposals for extending standard Petri nets to include time. Ramchandani (1974) proposed associating delays with transitions. Merlin (1974) proposed using two values, Min and Max times, to define a range of delays for each transition. This approach has also been used by Berthomieu and Menasche (1983). Sifakis (1977) proposed instead associating the delays with places. Coolahan and Roussopoulos (1983) employed an approach similar to Sifakis. Associating delays with places does not increase the power of the model, but does retain the instantaneous firing feature of the basic Petri net model. In fact, transition delays and place delays are equivalent since one can be translated into the other. Razouk (1983) has proposed using firing times along with enabling times. In his model, the tokens are absorbed by the transition after the enabling time has elapsed and do not reappear on the output places until after the transition finishes firing (i.e. after the firing time has elapsed). This model is less flexible than the Merlin and Farber model, but does make performance analysis easier.

Since our goal is not performance analysis using known times but the derivation of timing constraints, we have chosen to use the Merlin and Farber model. Tokens are allowed to remain on the input places during the transition delay so the model retains the instantaneous firing feature of untimed Petri nets while also providing a very flexible modeling tool.

A Time Petri net (TPN) is a Petri net, i.e. it is composed of a set of places  $P$ , a set of transitions  $T$ , an input function  $I$ , and an initial marking  $\mu_0$  along

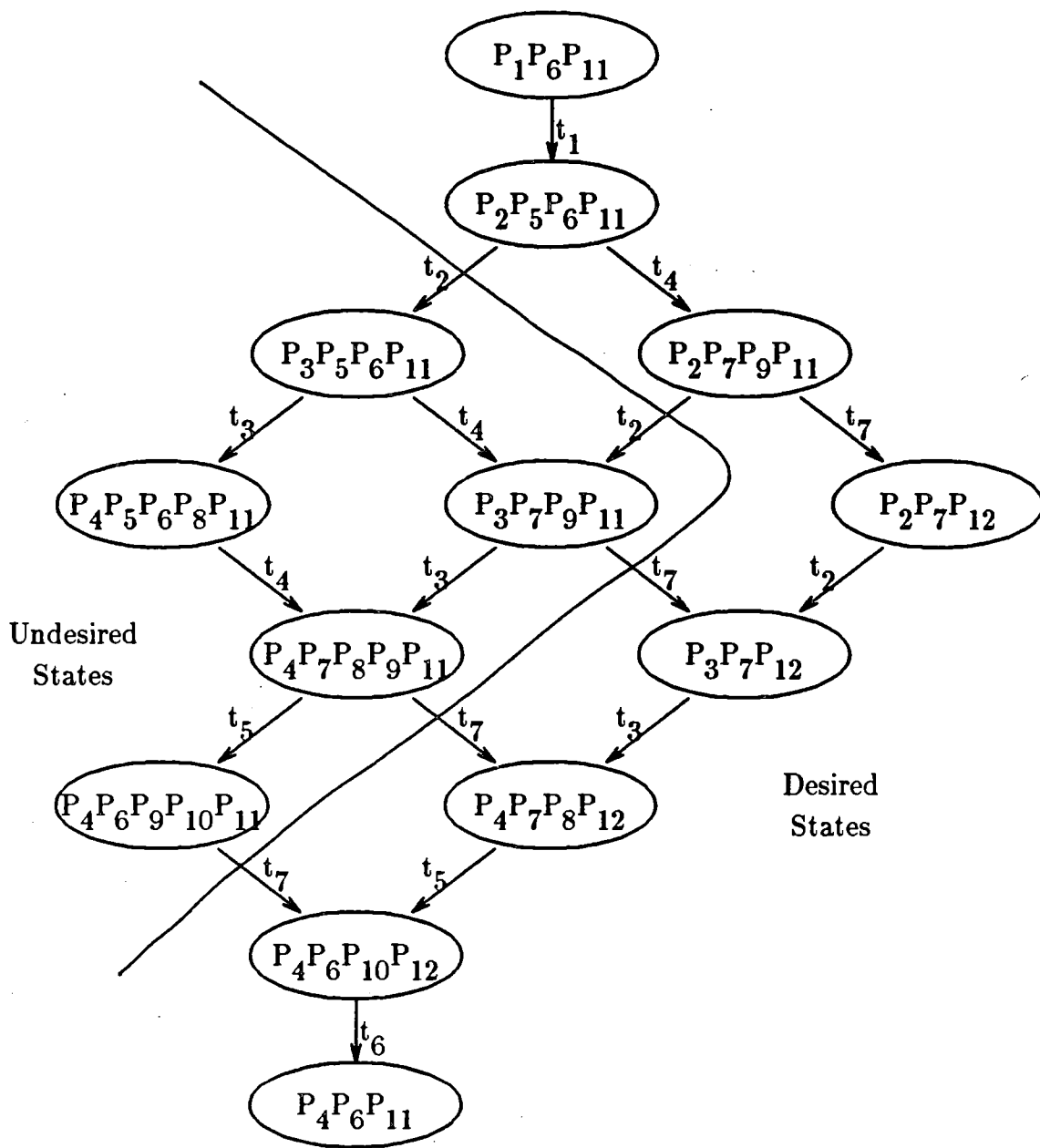


Figure 3. Reachability Graph for Figure 1

with the added *firing time functions* Min and Max. The firing time functions specify the conditions under which a transition may fire. Formally, this is written:

**Definition:** A *Time Petri net structure*,  $\Phi$ , is a seven-tuple,  $\Phi = (P, T, I, O, \text{Min}, \text{Max}, \mu_0)$ .

$P = \{p_1, p_2, \dots, p_n\}$  is a finite set of *places*,  $n \geq 0$ .

$T = \{t_1, t_2, \dots, t_m\}$  is a finite set of *transitions*,  $m \geq 0$ . The set of places and the set of transitions are disjoint,  $P \cap T = \emptyset$ .

$I: T \rightarrow P^\infty$  is the *input function*, a mapping from transitions to bags of places.

$O: T \rightarrow P^\infty$  is the *output function*, a mapping from transitions to bags of places.

Min and Max are the *min time function* and *max time function*, respectively, where

$\text{Min}: T \rightarrow R$  and  $\text{Max}: T \rightarrow R$ ,  $R$  is the set of non-negative real numbers and

$\text{Min}_i \leq \text{Max}_i$  for all  $i$  such that  $t_i \in T$ .

Finally,  $\mu_0: P \rightarrow N$  is the *initial marking* for the net where  $N$  is the set of non-negative integers.

**Definition:** A transition is *firable* at time  $\tau$  if and only if it has been continuously enabled during the interval  $\tau - \text{Min}(t_i)$  to  $\tau$ . The firable transition may fire at any time  $\tau$  for  $\text{Min}(t_i) \leq \tau \leq \text{Max}(t_i)$ . A transition must fire at time  $\tau$  if it has been continuously enabled during the interval  $\tau - \text{Max}(t_i)$  to  $\tau$ .

**Definition:** The state of the net  $\sigma$  consists of the tuple  $(\mu, E)$  where  $\mu$  is the marking and  $E$  is the remaining enabling time vector.  $E$  is a function of a set of tuples of real numbers  $R$ ,  $E: (R \times R) \rightarrow (R \times R)$

An excellent description of the next-state function for Time Petri nets can be found in Berthomieu and Menasche (1983). Added complexity over the untimed Petri net arises because of the continuous nature of time. Since transitions may fire at any time in their allowed interval, the states have in general an unbounded number of successors. Berthomieu and Menasche solve this problem by defining state classes which consider the set of all states reachable from the initial state by a given sequence of transitions.

Note that the Time Petri net is equivalent to a standard Petri net if all Min times are 0 and all Max times are set to  $\infty$ . Also note that the markings of the states of the Time Petri net reachability graph will be equal to or a subset of the

markings of the equivalent untimed Petri net. This is true since the enabling rules for the time Petri net are the same as for a Petri net. The difference lies in the additional restrictions placed on the firing rules. Thus adding timing may restrict the set of possible markings, but will never increase it. Since we are basically interested in determining worst cases (including the potential effects of having timing failures), much of our analysis will involve deriving the untimed reachability graph and then determining 1) the timing constraints of the final system necessary to avoid high-risk states, and 2) the run-time checks, e.g. watchdog timers, needed to be used to detect critical timing failures.

### Eliminating High-Risk States from the Design

A mishap is an unplanned event or series of events that results in death, injury, illness, or damage to or loss of property or equipment. Mishaps can be classified as to severity from catastrophic to negligible.

**Definition:** A *hazard* is a set of conditions within a state from which there is a path to a mishap. A state  $\sigma$  is hazardous if and only if there exists a mishap state  $\sigma_m$  and a sequence of transitions  $s \in T^*$  such that  $\delta^*(\sigma, s) = \sigma_m$

Hazards can be classified according to the severity of any possible resulting mishap. For simplicity we will divide hazards into two groups -- high-risk and low-risk -- where high-risk hazards can lead to catastrophic (unacceptable) losses. Of course more categories can and often are used. It is important to note that in many, if not most, realistic systems it is impossible to completely eliminate risk. The goal instead is to design a system with "acceptable risk."<sup>†</sup>

To show that a system is *safe*<sup>††</sup> or *low-risk*, it is necessary to first ensure that given that the specifications are correctly implemented, no mishaps will result. Second, the risk of faults or failures leading to a mishap must be eliminated or minimized. In this section we discuss how to identify and eliminate high-risk hazards which have been designed into the system. The next section

---

<sup>†</sup>What is acceptable risk is often determined by appropriate government licensing agencies. For example, mishaps have been defined by the NRC for all nuclear systems. If not predetermined by law, the definition and categorization of mishaps as to severity must be done in the early stages of the system design.

<sup>††</sup>Because the term "safe" has a specific meaning in Petri net theory (a place is safe if it never contains more than one token), we will use the term "low-risk" where necessary to avoid confusion.



will treat the problem of failures.

Creating the reachability graph allows the designer of a system to determine if the system design can "reach" any high-risk states since it determines all possible states that the system can reach from the initial state by any legal sequence of transition firings. However, this may well be impractical due to the size of the reachability graph for a complex system. In the rest of this section, we describe techniques which may allow the design to be analyzed for safety without producing the entire reachability graph.

The states of a reachability graph can be separated into two disjoint sets: States from which it is possible to reach high-risk and possibly also low-risk states and those from which it is possible to reach only low-risk states.

**Definition:** A state (marking)  $\mu_c$  is a *critical state* if and only if

- a)  $\mu_c \in$  low-risk states and
- b) there exist two sequences of transitions,  $s_1$  and  $s_2$ , a  $\mu_i$  and a  $\mu_j$  such that  $\delta^*(\mu_c, s_1) = \mu_i$  and  $\delta^*(\mu_c, s_2) = \mu_j$  where  $\mu_i \in$  high-risk states and  $\mu_j \in$  low-risk states.

If a high-risk state is reachable, then there must be a critical state on the path from the initial state to the high-risk state (this includes the possibility that the critical state is the initial state). Otherwise, the design needs to be completely redone from scratch since all executions result in high-risk states.

To eliminate hazards, it is not necessary to produce the entire reachability graph but only to determine the critical states and to disallow the unwanted transition in each case. Some of our techniques are conservative, i.e. in order to reduce the large amount of computing to produce the entire graph, a larger number of critical states may be identified than actually exist. But note that it does no harm to eliminate a hazard which never existed. Also, as will be seen in the next section when failures are discussed, eliminating a non-existent path may have the effect of eliminating or lessening the possibility of mishaps caused by run-time failures and faults.

One way to locate critical states without necessarily producing the entire reachability graph is to start with the set of high-risk states and to work backward to determine if they are reachable from the initial state. This approach is useful when the goal of the analysis is only to prove that the system cannot reach certain hazardous states. This is often a requirement for safety-critical systems, e.g. see MIL-STD-882b. Fault tree analysis is a similar technique used for the same purpose [Vesely *et. al.* (1981)]. The backward approach is itself practical only if one considers a relatively small number of high-risk states. This has been

found to be adequate in practice [Vesely *et. al.* (1981)]. Note that the concern here is not with correctness, but with system safety. That is, a system is "safe" if it is free from mishaps even if it also does not accomplish its "mission" or functional objectives.

To determine if a state can be reached using backward reachability graphs, it is necessary to temporarily ignore timing constraints. The procedure is to first construct the *inverse* untimed Petri net.

**Definition:** The *inverse* Petri net,  $\Phi^{-1}$  for a Petri net  $\Phi=(P,T,I,O)$  is defined by interchanging the input and output functions,  $\Phi^{-1}=(P,T,O,I)$ .

A reachability graph is then constructed using the inverse Petri net and the high-risk state as the initial marking. If the original initial state is reachable, then the mishap may be possible.

**Theorem:** A high-risk state  $\sigma_m$  is in the reachability set  $R(\Phi, \sigma_0)$  if and only if given an initial state  $\sigma_0$ ,  $\sigma_0 \in R(\Phi^{-1}, \sigma_m)$ .

The proof can be shown by induction on the sequence of transition firings. By definition if  $\mu = \delta(\mu', t)$  then  $\mu' = \delta^{-1}(\mu, t)$ . This allows the sequence of transitions from  $\sigma_0$  to  $\sigma_m$  to be traversed in reverse order.

Even though a high-risk state is reachable in the untimed Petri net, it may not be reachable when time constraints are considered. Two approaches are possible. The first is to use the time constraints and work forward from the initial state to determine if the timing constraints have eliminated this path from the timed reachability graph. The other is to assume the worst and just modify the design to ensure that the path is eliminated.

This backward approach is only helpful if the resulting reachability graph is smaller than the original. If the state is reachable, then the backward reachability graph can never be larger than the original reachability graph. Unfortunately, if the high-risk state is not reachable, it is possible for the backward reachability graph to actually be larger than the original graph and even to be infinite. Therefore, again it may be impractical to generate the entire backward reachability graph.

But if the goal is to ensure that high-risk states can never be reached, it is possible to simply work backward to the first "critical" state (in this case to a state in the reachability graph which has two successors) and to use design techniques such as those outlined below to ensure that the bad path is never taken. It is unimportant as to whether this path is actually reachable since eliminating the possibility of a mishap which would not have occurred does no harm. It is

also unimportant if this is truly a critical state as defined above (one path leads to low-risk states) since if the uneliminated path also leads to a mishap, this will be determined in a later step, and this second path will also be eliminated.

The following describes the details of the algorithm to identify and eliminate critical states:

Put initial set of high-risk conditions into  $S = \text{states\_to\_process}$

while  $S$  is not empty

do

let  $c$  be one of  $S$ ;

if  $c$  is a subset of the initial state then

high-risk state reachable and need to redesign

else

do {work backwards to critical states}

{determine which transitions are enabled}

for each transition  $t \in T$

do

let  $R = O(t) \cap c$ ;

$TE = O(t) - R$ ;

$SE = c - R$ ;

if  $R \neq \emptyset$  then { $t$  is enabled and we need to

{generate the corresponding next backward states}

$\text{Next\_back\_states} = \text{Next\_back\_states} \cup \delta^{-1}(R \cup TE \cup SE, t)$ ;

od

for each next\_back\_state  $b$

do

$\text{Forward\_states} = \text{set of states } \delta(b, t)$

$\text{Other\_states} \leftarrow \text{Forward\_states} - [\text{Forward\_states} \cap \{S \cup \text{Next\_back\_states}\}]$

case  $b$

$b \in \text{states\_considered}$  : exit;

$b$  is illegal according to system invariants : exit;

$b$  is high risk : add  $b$  to  $S$ ;

$b$  is low-risk and there exists a  $f \in \text{other\_states}$  such

that  $f$  is low-risk {therefore  $b$  is potentially critical}: add  $b$  to set of critical states;

else { $b$  is low-risk but not critical - necessary to go

backwards again}

add  $b$  to  $S$ ;

esac

```
    od
    move C from S to states_considered;
    augment design by eliminating bad transitions from critical states;
  od
end while
```

The above algorithm starts with the set of high-risk conditions. For each member of this set, the immediately prior state or states are generated. Each of these "one-step-backward" states is then examined to see if it is a potentially critical state and can be used to eliminate one path to the high-risk state. Note that we are not dealing with complete states but only with partial states. That is, some conditions in the state are unimportant as far as risk goes. Furthermore, we do not know what the complete final states are. Therefore there may be some "don't care" places in each state which are determined in the process of executing the algorithm. Finally, we only need to look forward one step from each potentially critical state in order to label it as critical (i.e. there exists a next-state which is low-risk). This is because if this path also leads to a high-risk state, then it will be eliminated by the algorithm in a later step.

Using the train example again, figure 3b shows the partial graph generated by the algorithm for the high risk state where the train is approaching ( $P_3$ ), the gate is up ( $P_{11}$ ), and any other "don't care conditions" (denoted by the "\*") may also hold. Propagating this state backwards, we reach the initial state, impossible states, and critical states. From this we derive the information that in order to avoid the high-risk state, the design must be modified to ensure that transition  $t_3$  has priority over transition  $t_{11}$  and that transition  $t_6$  has priority over transition  $t_1$ .

When a critical state is identified, it is necessary to modify the Petri net in some way to ensure that the good path is always taken, i.e. that another transition always is performed before or has priority over the critical transition.<sup>†</sup>

There are many possible ways of modifying the system design to eliminate the high-risk states. One common approach is to use an interlock. Interlocks are used to ensure correct sequences of events. An example of a hardware interlock is an access panel or door to equipment where a high voltage exists. Software

---

<sup>†</sup>By requiring that a transition  $t_i$  always has priority over a transition  $t_j$  in all situations may be more strict than absolutely necessary but this is true of most safety devices and is one reason why safety occasionally conflicts with other system qualities such as performance.

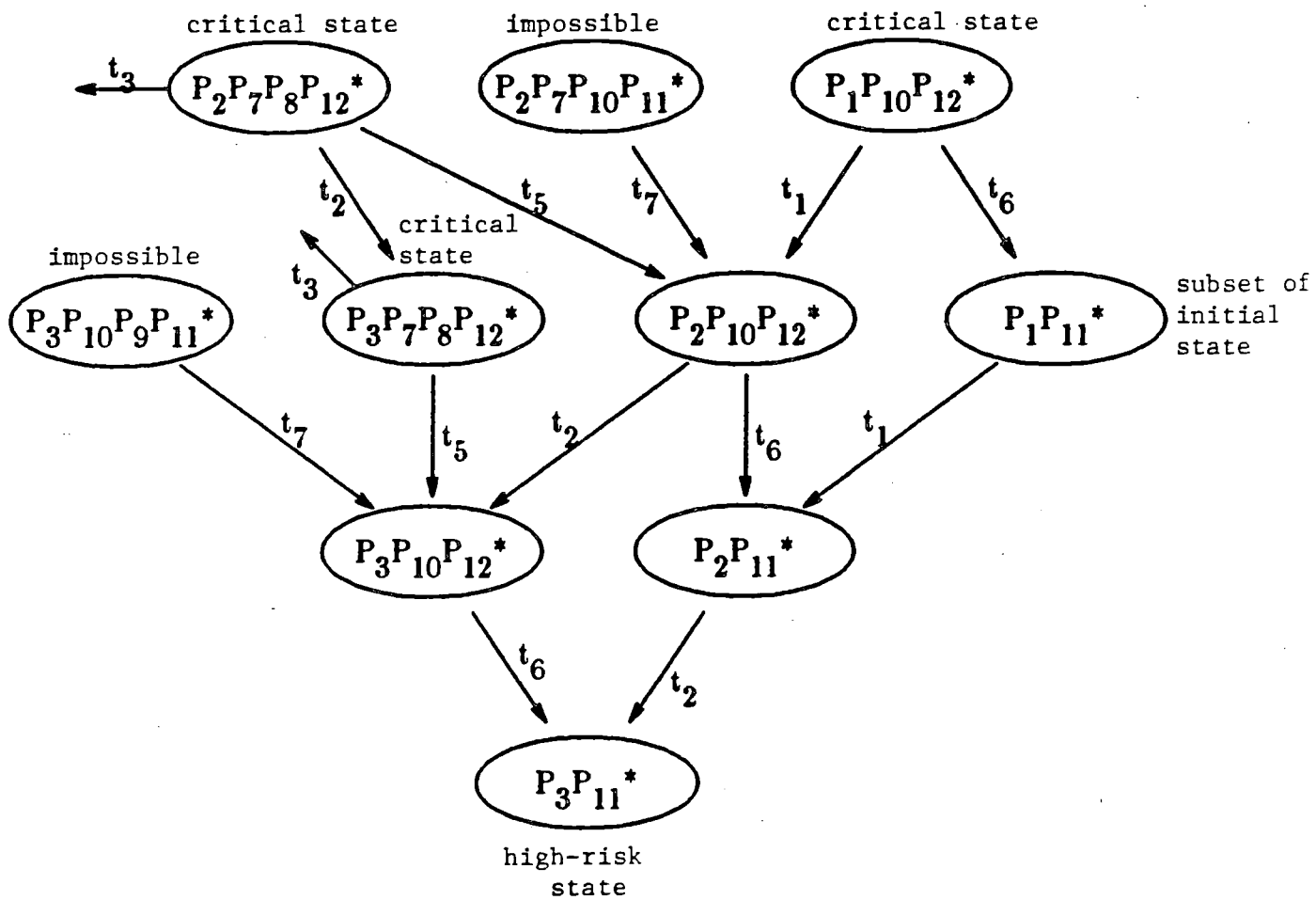


Figure 3b. Example of Critical State Algorithm

interlocks include monitors and batons. To model an interlock in a Petri net, assume that  $t_i$  is the desired transition, while  $t_j$  is the undesired transition. It is possible to force the system always to take the desired path (i.e. to eliminate the undesired path from the reachability graph) by making the following changes to the two transitions in the Petri net. Add a new place (the interlock I) to the output bag of  $t_i$  and to the input bag of  $t_j$ . This ensures that transition  $t_i$  always has priority over transition  $t_j$ . There may be multiple desired transitions and an interlock must be applied to each. See figure 4a for an example.

The above type of interlock is used to ensure that one event always precedes another event (e.g. a baton in software). Another type involves ensuring that an event does not occur while a condition is true. This is implemented in the Petri net by using a locking place (see figure 4b). This corresponds to a critical section in software.

In the train example, an interlock can be added between  $t_7$  and  $t_2$  (see figures 4c and 4d) in order to eliminate the high-risk states. The interlock is included within the computer-controller, but alternatively it might have been part of the hardware. One physical implementation of such an interlock might be a computer-controlled warning signal for the train.

Another way to ensure that one transition will always fire when both are enabled is to enforce timing constraints or timing conditions in the designed system. In order to ensure that a transition  $t_j$  (which leads to the high-risk state) does not fire whenever  $t_i$  and  $t_j$  are both enabled (i.e. the high-risk state is eliminated from the reachability graph), the following timing constraint must be enforced: the maximum time that it may take for the higher priority transition ( $t_i$ ) to fire must be less than the minimum time for the lower priority transition ( $t_j$ ) to become enabled and to fire. Each of these time quantities must be the total time that the enabling conditions have been met, not just the individual transition time limit.

One method of determining these quantities is to use the reachability graph to find the maximum (minimum) valued path leading to the transition which has the required conditions continually enabled. In the system modeled in figures 1 and 3, the desired goal is to have condition  $P_{12}$  occur before condition  $P_3$ . In terms of the reachability graph this means that when in state  $P_2P_5P_6P_{11}$  or  $P_2P_7P_9P_{11}$ , transition  $t_2$  must not be firable. In the first case, the constraint necessary for  $t_4$  to fire before  $t_2$  is simply that  $\text{Min}(t_2) > \text{Max}(t_4)$ . For the second case it is a bit more complicated since firing  $t_1$  results in  $t_2$  being enabled. The constraint in this case is  $\text{Min}(t_2) > \text{Max}(t_7) + \text{Max}(t_4)$ .

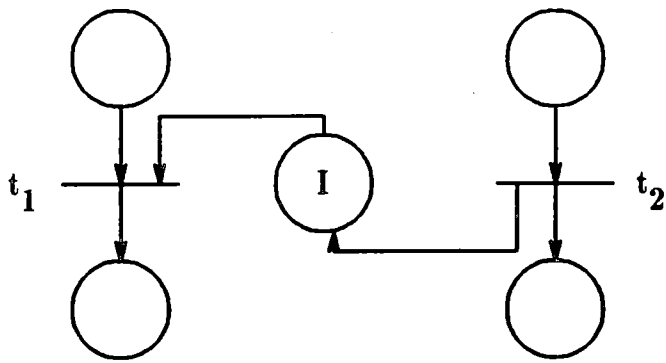


Figure 4a. Interlock

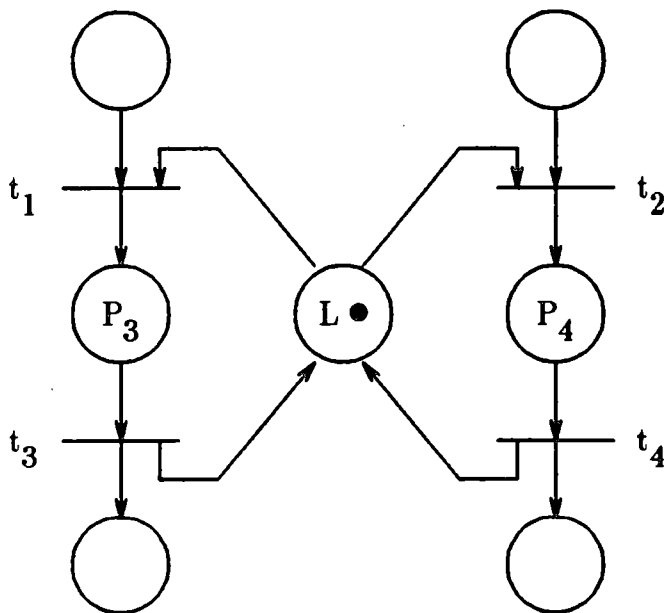


Figure 4b. Locking Place

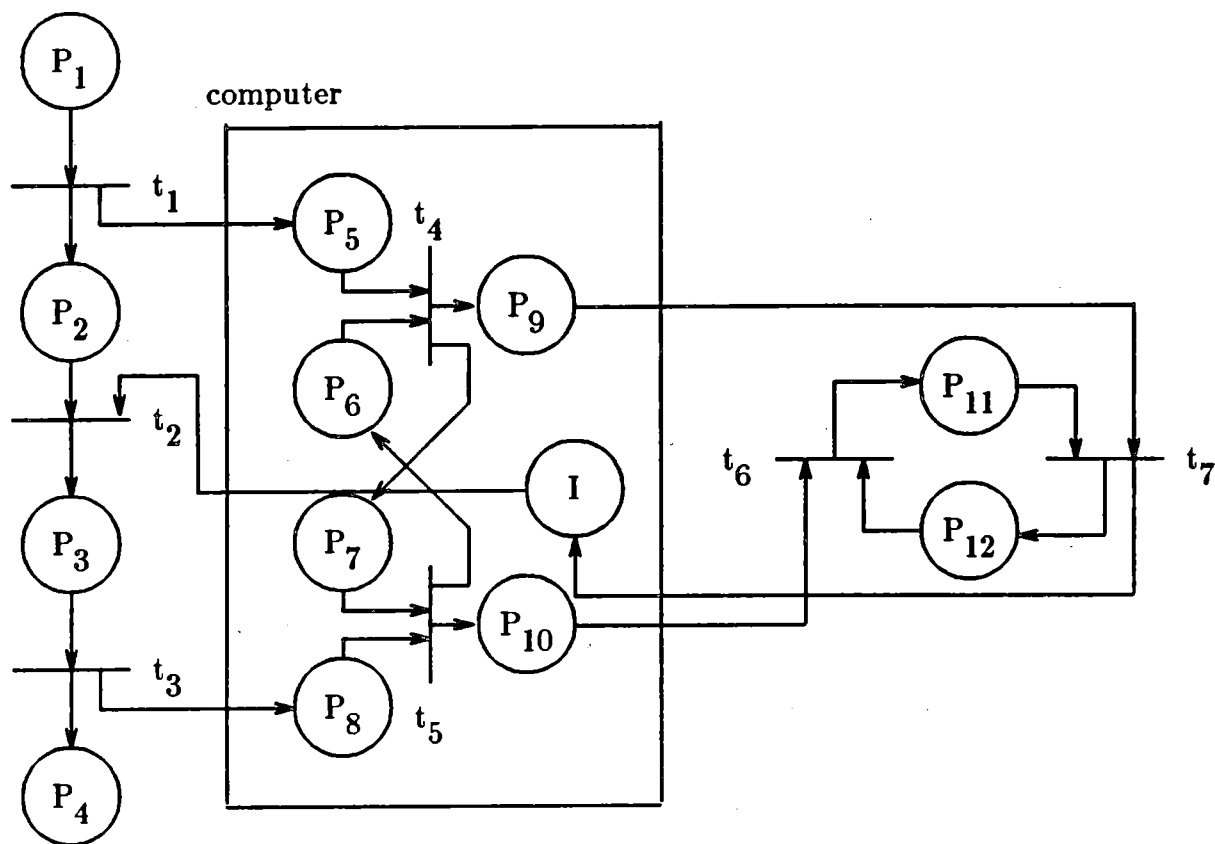


Figure 4c. A Petri Net Graph with an Interlock

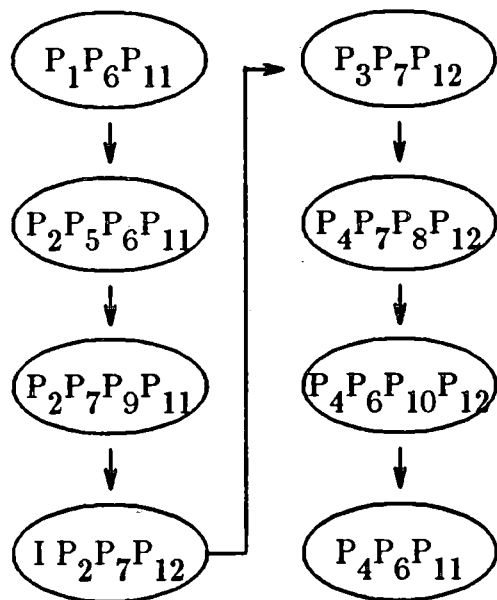


Figure 4d. New Reachability Graph



Timing constraints are enforced in systems by either verifying that the design makes it impossible for the constraint to be violated or by using watchdog timers and other devices to determine when the constraint is about to fail and to insert recovery techniques into the system design (either software or hardware). An example is shown in the next section.

### **Adding Failures to the Analysis**

Once the design is determined to have an acceptable level of risk, run-time faults and failures must be considered. Designing for fault tolerance and safety requires being able to model failures and faults and to analyze the resulting model. In order not to cause confusion by providing our own definitions of terms like "failure," we have tried to use standard definitions. However, there appears to be some disagreement, so in this paper we take definitions from Kopetz (1982), Laprie (1982), and Anderson and Lee (1982). A failure is defined as an event while a fault is a state. A failure always results in a fault and is called a fault-starting event. The fault remains in the system until the occurrence of a terminating event for this fault. In the time domain, failures can be characterized by their frequency of occurrence within a given time interval as either singular (once), intermittent (sometimes), or persistent (always). Laprie (1982) further separates failures into benign and catastrophic.

In this paper, we are concerned with control failures. Control failures include:

- a required event that does not occur
- an undesired event
- an incorrect sequence of required events
- two incompatible events occurring simultaneously
- timing failures in event sequences
  - exceeding maximum time constraints between events
  - failing to ensure minimum time constraints between events
  - durational failures (i.e. a condition or set of conditions fail to hold for a particular amount of time)

Each of these types of failures must be able to be modeled in the Petri net. Merlin and Farber (1976) modeled failures in Petri nets as a loss of token or generation of a spurious token. Azema and Diaz (1977) took a similar approach. This was appropriate since Merlin's goal was to analyze failures in communication systems where the primary type of fault is the loss of a message due to

failure of the underlying communication medium. However, when dealing with analysis of failures in more general situations, it is often useful to be able to determine the state that a system is in after the failure has occurred (i.e. the fault). For example, if a token is lost when the system is in a state where a particular bit is 1, it is important to know whether the failure results in a "stuck at 1" state or a "zero" state for the bit. This is because a fault remains in the system until a terminating event for the fault (the faulty condition is no longer true or loses its token). Because of the faulty state or condition, it is possible for further failures to occur [Kopetz (1982)] which cause further faults. Thus the type of fault which results from the failure must be included in the model in order to analyze the consequences of failures on the system (and thus to differentiate between high and low cost failures). For analysis and readability purposes, it is also useful to model failure events in a different way than normal, expected events.

For these reasons, we introduce a new type of transition, a *failure transition* which acts like other transitions but is denoted by a dotted bar and a *fault condition* which is denoted by a dotted circle.<sup>†</sup> For a Petri net,  $\Phi$ , the set of transitions becomes  $T = T_L \cup T_F$  where  $T_L$  are legal transitions and  $T_F$  are failure transitions and  $T_L \cap T_F = \emptyset$ . Similarly, the set of places is now  $P = P_L \cup P_F$  where  $P_L$  are legal places and  $P_F$  are faults and  $P_L \cap P_F = \emptyset$ . Examples of modeling some of the above types of control failures can be found in figure 5. The failure transitions shown are infinitely fire-able. To make analysis practical, a place which acts as a counter can be added to the failure transition. The number of tokens initially contained in this place controls the maximum number of times the transition (failure) can fire. Realistically, most systems are designed for a maximum number of failures and the tokens in the counter are the Petri net equivalent of this ceiling value.

We now have two types of states: faulty states and legal states.

**Definition:** A state  $\sigma$  is a *legal state* if and only if there exists a path in the failure reachability graph from the initial state  $\sigma_0$  to  $\sigma$  which contains only legal transitions, i.e. if  $\sigma_0$  is the initial state, and there exists a sequence of legal transitions  $s \in T_L^*$  such that  $\delta^*(\sigma_0, s) = \sigma$ .

<sup>†</sup>Merlin actually includes failure transitions in his reachability graph (which he calls the error token machine), but does not put them in the Petri net itself.

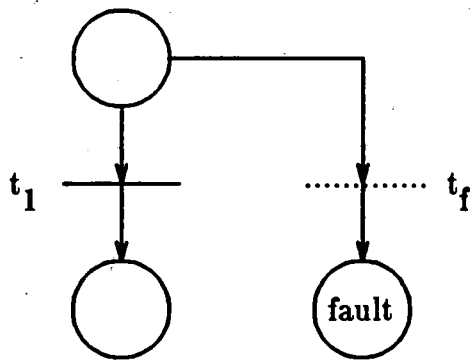


Figure 5a. Desired Event  $t_1$  Does Not Occur

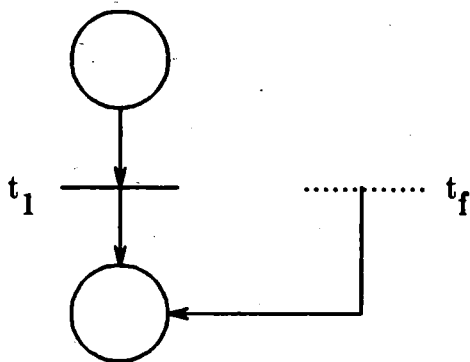


Figure 5b. Undesired Event  $t_1$  Occurs

**Definition:** A state  $\sigma$  is a *faulty state* if and only if every path  $P$  to  $\sigma$  from the initial state  $\sigma_0$  contains a failure transition i.e. for every sequence  $s \in T^*$  where  $\delta^*(\sigma_0, s) = \sigma$  there exists a  $t_f$  such that  $t_f \in T_F$  and  $t_f \in s$

Once failures are included in the model, it is necessary to decide what qualities of the design are important to analyze with respect to control failures. Three such qualities are control fault tolerance, recoverability, and fail-safety. Control fault tolerance implies that a system continues to function correctly (i.e. to provide the service required by its specification) in the presence of component failure. Recoverability implies that a system continues to provide service although the service may be (temporarily) degraded (i.e. may not satisfy *all* the requirements of the specification). A system is fail-safe if component faults do not lead to a catastrophic system failure (mishap) although the system may not provide any service except that required to prevent the catastrophic failure. Each of these qualities can be defined in terms of Petri nets as follows:

**Definition:** A process  $P$  is *recoverable* if after the occurrence of a failure, the control of the process is not lost, and in an acceptable amount of time, it will return to normal execution. Formally, a process  $P$  is recoverable from a failure  $t_f \in T_F$  if and only if in the failure reachability graph (FRG):

Let  $\Sigma_F$  be the set of faulty states and let  $\Sigma_L$  be the set of legal states

1) the number of faulty states is finite,

$$\text{cardinality}(\Sigma_F) < \infty$$

2) there are no terminal faulty states,

for all  $\sigma \in \Sigma_F$ ,  $\sigma$  is firable

3) there are no directed loops including *only* faulty states,

there does not exist a sequence  $t_1 \dots t_n$  in the FRG such that for  $\sigma_i \in \Sigma_F$ ,

$$\delta(\sigma_i, t_i) = \sigma_{i+1} \text{ for } i=1..n-1 \text{ and } \sigma_1 = \sigma_{n+1}$$

4) the sum of the maximum times on all paths from the failure transition to a correct state is less than a pre-defined acceptable amount of time.

For every path  $P(t_1, \dots, t_n)$  from  $\sigma_1 \in \Sigma_F$  to  $\sigma_2 \in \Sigma_L$ ,

$$\sum \text{Max}(t_j) < T_{\text{acceptable}} \text{ for } j = 1..n$$

This definition is similar to that of Merlin and Farber (1976), but they allow any finite amount of time to return to normal execution. For many real-time

systems, timing constraints are more strict than this. Thus doing nothing for a certain amount of time can be as dangerous under certain conditions as performing an incorrect action even though control is ultimately restored.

**Definition:** A string A is a *subsequence* of string B if and only if A can be obtained from B by deleting zero or more elements of B.

**Definition:** A process P is *fault-tolerant* for a control failure  $t_f \in T_F$  if and only if a) it is recoverable and b) a correct behavior path is a subsequence of every path from the initial state to any terminal state. A correct behavior path is a path in the FRG from the initial state to final state which contains no failure transitions, i.e. a sequence of transitions  $t_1 \dots t_n \in T^*$  such that for all i,  $t_i \in T_L$  and  $\delta(\sigma_{i-1}, t_i) = \sigma_i$ , for  $i=1..n$ ,  $\sigma_n$  is not firable

Note that for nonterminating or cyclic processes,  $\sigma_n$  may not be a terminal state but may instead be the initial state.

**Definition:** A system is *fail-safe* if and only if all paths from a failure F in the FRG contain only low-risk states.

i.e. for all states  $\sigma_f$  and sequences  $s_1$  such that  $\delta^*(\sigma_0, s_1 F) = \sigma_f$   
there does not exist a sequence  $s_2$  and state  $\sigma_h \in \text{high-risk states}$  such that  $\delta^*(\sigma_f, s_2 F) = \sigma_h$

Note that the system may never get back to a legal state.

The above definitions can be extended to include the possibility of n failures, thus a system, for example, may be n-fault tolerant, n+1-recoverable, and n+2-fail-safe.

Two analysis approaches are possible. The first is to determine, perhaps through past experience, which failures are most likely, and then to create the resulting Failure Reachability Graph (FRG) and analyze it for the above properties. This may be very costly (and possibly impractical) for complex systems with many possible failure modes. Also, in software it is difficult to determine directly which failures are the most likely.

An alternative approach is to take the safety viewpoint and consider only those failures with the most serious consequences. Since this is the requirement of most safety certification programs, there is a practical application for this type of analysis. In this approach, single-point failures and failure sequences which can lead to high-risk states are determined through the analysis after which the design can be augmented with fault-detection and recovery devices to minimize the risk of a mishap. If risk cannot be lowered sufficiently through these devices

(e.g. there is an unacceptable probability they will fail or there are uncontrollable variables such as human error involved), it is also possible to add additional safety devices to the design. For example, the designer may add hazard-detection and risk-minimization mechanisms which attempt to ensure that if a hazardous state is reached, the risk will be eliminated or minimized by fail-safe techniques which change the state to a no-risk or lesser-risk state while at the same time minimizing the exposure time of the hazard.

As an example of the process, consider the Petri-net model in the previous examples. If interested in failures which could result in high-risk states (e.g. the train is approaching,  $P_3$ , and the gate is up,  $P_{11}$ ), a backward reachability graph can be constructed (figure 6b). The high-risk state is not reachable from the regular Petri net, but examination of the reachability graph in figure 6c shows that three single failures (each by themselves) would allow the high-risk state to be reached, i.e. a failure transition  $f_2$  which takes a token from  $P_2$  and puts one in  $P_3$ , a failure transition  $f_6$  which does the same for  $P_{12}$  and  $P_{11}$ , and a failure transition  $f_5$  which involves an erroneous generation of a token in  $P_{10}$ . Failure transition  $f_2$  is a human failure where the train ignores the warning signal. Transition  $f_6$  is a gate failure which results in a premature gate raising. The last failure,  $f_5$  could be caused by a spurious signal from the controlling computer. Normally, the designer would now include standard failure detection mechanisms in the design along with recovery procedures.

Failure transition  $f_5$  in figure 6a was chosen as the basis for the fault tolerance mechanism shown in figure 7. This failure models a spurious output signal from the computer. The number of tokens in  $P_{14}$  represent the maximum number of failures that can occur during analysis. The analysis performed here is for at most one failure. Transitions  $R_1$  and  $R_2$  are used for fault detection and subsequent recovery. After a failure, the system can be in two possible situations depending on the current state of the gate. If the gate is up then one response to a spurious up signal is to ignore it (shown in transition,  $R_2$ ). The enabling conditions are  $P_{11}$  (gate up) and  $P_{10}$  (signal from the computer).

The second possibility is the safety critical situation. In this case a train in approaching, the gate is down, and the erroneous signal is given to raise the gate. In order to detect the situation redundant information must be contained in the system. The model has an internal "view of the world" contained in  $P_6$  and  $P_7$  which correspond directly to the actual conditions  $P_{11}$  and  $P_{12}$ . Fault detection is accomplished by checking to see if  $P_7$  and  $P_{11}$  occur at the same time. If so, there is a discrepancy between the real world and the internal state.

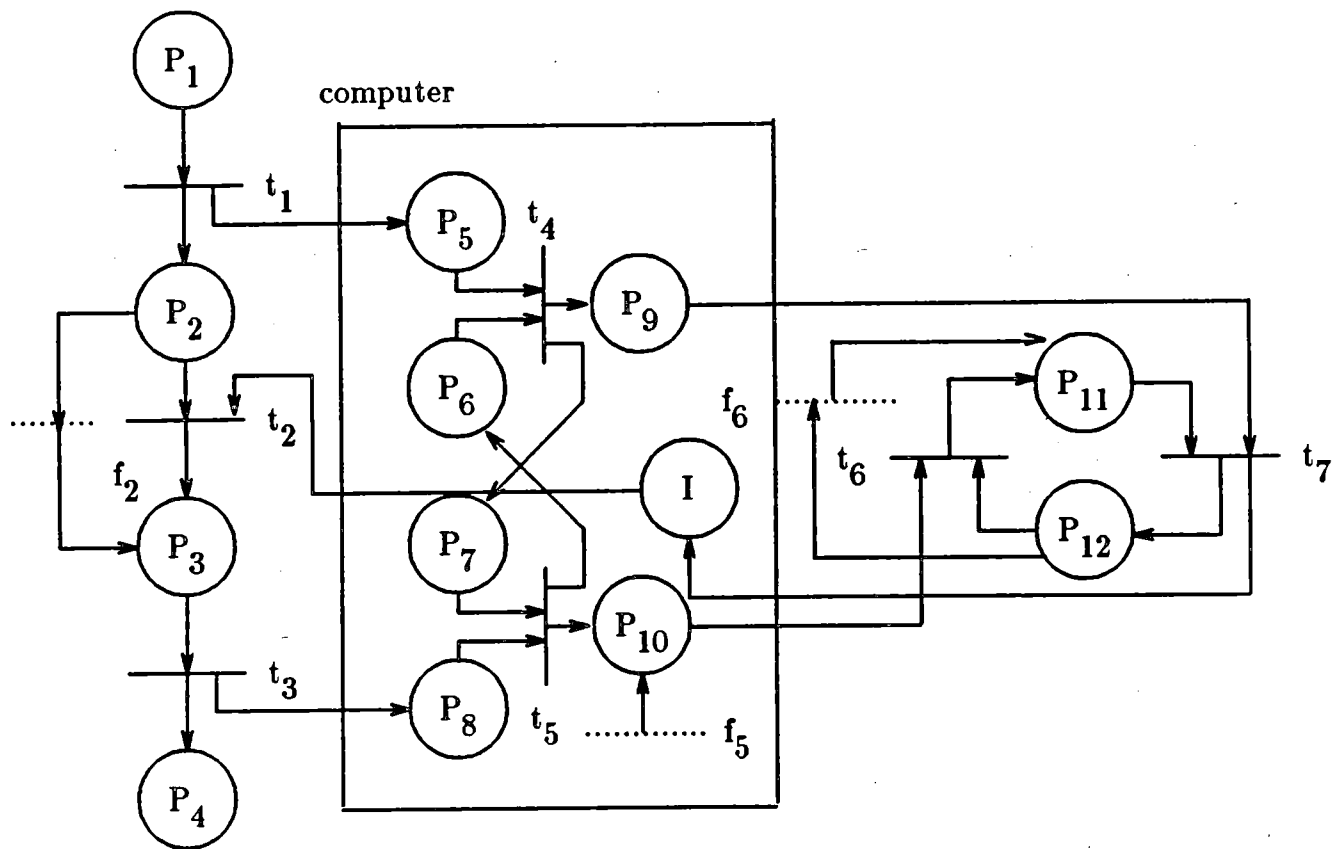


Figure 6a. A Petri Net Graph with Failures

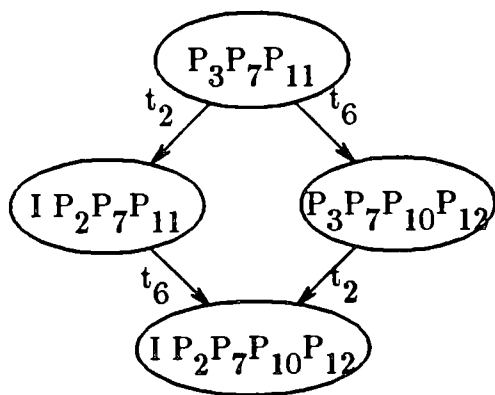


Figure 6b

Backwards Reachability Graph

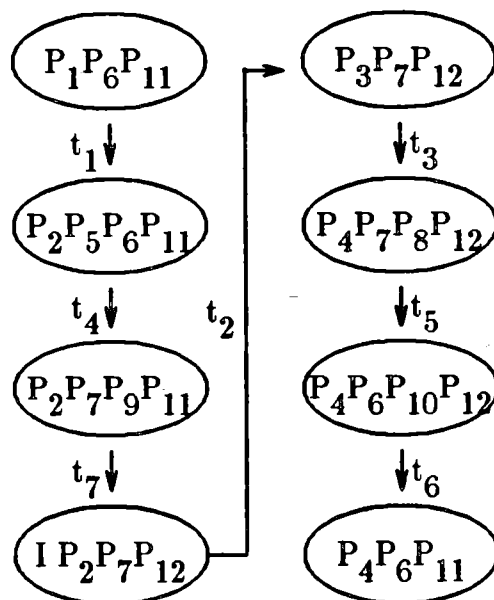


Figure 6c

Reachability Graph for Figure 6a

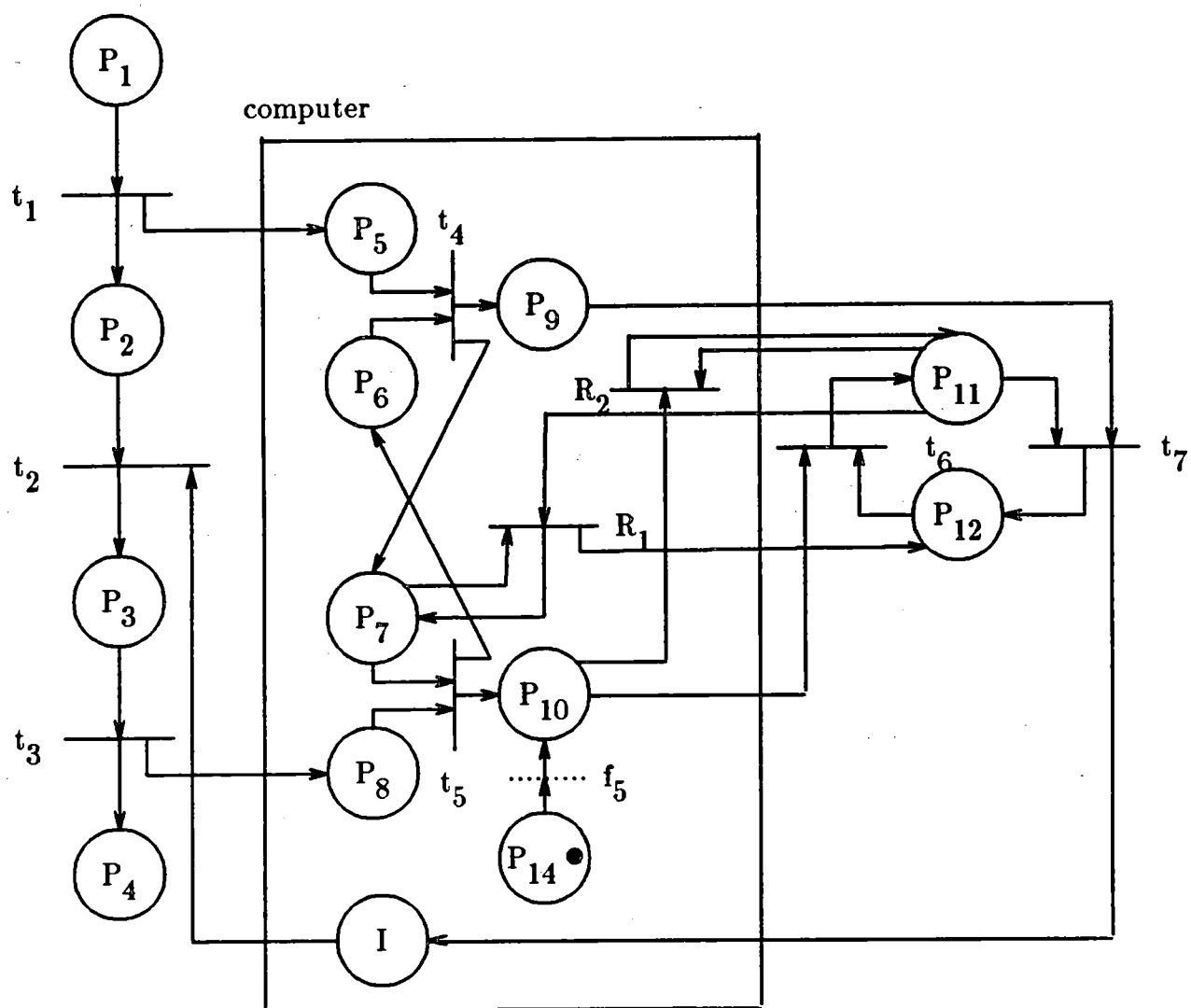


Figure 7. A Petri Net Graph with Failure Transition and Recovery



Upon failure detection, there are several possible recoveries -- depending on which model is accepted as the true state of the system (i.e. is the computer state wrong or is the gate really up when it should be down). The safest solution is to assume the gate is up and lower it. This is the purpose of transition,  $R_1$ . Figure 8 shows the reachability graph for this net. From the untimed reachability graph we see that for the state labelled 4 (conditions  $P_2$ ,  $P_7$ ,  $P_9$ ,  $P_{11}$ , and  $P_{14}$ ), recovery is initiated when a failure has not occurred. Further investigation reveals that there is a point in time when the computer state is legitimately inconsistent with the actual world (after  $t_4$  has fired but before  $t_7$  fires). One solution is to put a time constraint on  $R_1$  such that the minimum time of  $R_1$  is greater than the maximum time of  $t_7$ . This forces failure detection to wait until a consistent state has been permitted.

In summary, analysis of the failure reachability graph with respect to the definitions of fault tolerant, recoverable, and fail-safe design will aid the designer in adding appropriate failure detection and recovery techniques to the system. When interested solely in a safety analysis, backward procedures can be used to determine which failures and faults are potentially the most costly and thus need to be augmented with fault tolerance mechanisms and also to determine where and how safety mechanisms should be used. This may be particularly useful for the software components of the system since it is difficult to determine which faults are most likely to occur and the potential number of failures to model may be very large. Furthermore, it is possible to treat the software at various levels of abstraction, e.g. only failures of the interfaces of the software and non-software components may be considered or more detailed failures of only those particular modules which are determined to be critical may be modeled.

## Conclusions

The use of Time Petri nets in design and analysis of safety-critical, real-time systems has been described and the basic model extended to allow modeling failures and faults. This allows the system to be analyzed for properties such as fault-tolerance and safety, to determine which functions are most critical and thus may need to be made fault-tolerant (assuming that it may be too costly to ensure complete fault-tolerance), to determine conditions which require immediate mitigating action to prevent accidents, to determine possible sequences of failures which can lead to accidents, etc. Thus it is possible to establish important properties during the synthesis of the design instead of using guesswork and costly *a posteriori* analysis (including formal analysis and testing).

State #	Places
1	$P_1 P_6 P_{11} P_{14}$
2	$P_2 P_5 P_6 P_{11} P_{14}$
3	$P_1 P_6 P_{10} P_{11}$
4	$P_2 P_7 P_9 P_{11} P_{14}$
5	$P_2 P_5 P_6 P_{10} P_{11}$
6	$P_1 P_6 P_{11}$
7	$P_2 P_7 P_{12} P_{14}^I$
8	$P_2 P_7 P_9 P_{10} P_{11}$
9	$P_2 P_7 P_9 P_{12} P_{14}$
10	$P_2 P_5 P_6 P_{11}$
11	$P_3 P_7 P_{12} P_{14}$
12	$P_2 P_7 P_{10} P_{12}$
13	$P_2 P_7 P_9 P_{10} P_{12}$
14	$P_2 P_7 P_9 P_{11}$
15	$P_4 P_7 P_8 P_{12} P_{14}$
16	$P_3 P_7 P_{10} P_{12}$
17	$P_2 P_7 P_{11}$
18	$P_2 P_7 P_{12}$
19	$P_2 P_7 P_9 P_{12}$
20	$P_4 P_6 P_{10} P_{12} P_{14}$
21	$P_4 P_7 P_8 P_{10} P_{12}$
22	$P_3 P_7 P_{11}$
23	$P_3 P_7 P_{12}$
24	$P_4 P_6 P_{11} P_{14}$
25	$P_4 P_6 P_{10} P_{10} P_{12}$
26	$P_4 P_7 P_8 P_{11}$
27	$P_4 P_7 P_8 P_{12}$
28	$P_4 P_6 P_{10} P_{11}$
29	$P_4 P_6 P_{11}$
30	$P_4 P_6 P_{10} P_{12}$

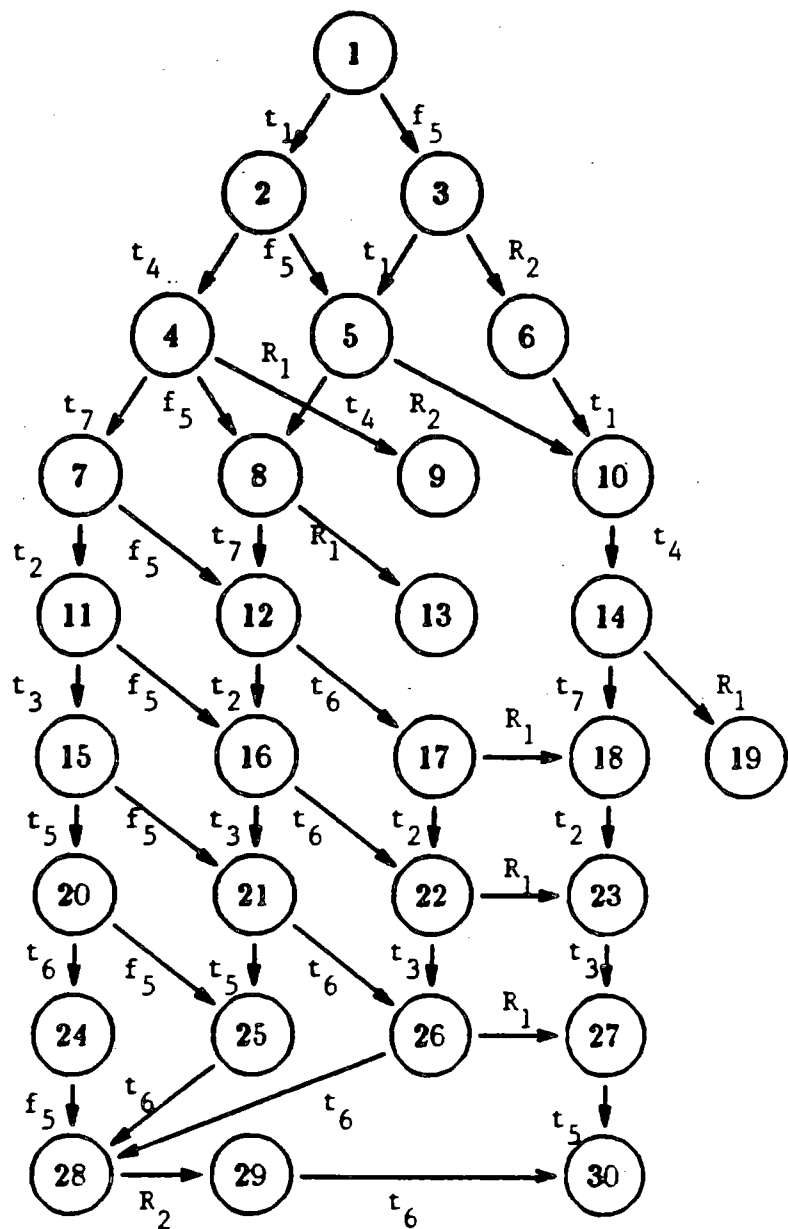


Figure 8. Reachability Graph for Figure 7

Unfortunately, Petri nets can be difficult to analyze. For general Petri nets, the reachability problem, though decidable, has been shown to be exponential time- and space-hard. Although this is not a necessary property of Petri net models (many important and real systems can be analyzed efficiently), it is a possible result when very complex systems are modeled. Some techniques which are useful even if the entire reachability graph is not completed have been presented in this paper. It is also possible to use the failure-enhanced Time Petri net model as the basis for a simulation in order to answer some of the same questions which could have been answered by the failure reachability graph. Finally, many real-time systems require the computer software to be written and tested before the hardware components have been completed. Since the time Petri net model is executable, the hardware parts can be used as a test bed for the software development process.

In this paper, only severity of hazards were considered and not the probability of the hazard occurring or of leading to a mishap. This is a pessimistic approach (i.e. all hazards are considered to have equally high probabilities). We are currently devising techniques to include probabilities in the analysis. This will enable the designer to use a more sophisticated definition of risk and to derive measurements for risk (and thus safety) from the model. This in turn can provide the information required by the designer to make difficult tradeoff decisions, e.g. what if there are two possible recovery methods, one of which is more likely to work but also has worse penalties in the event of failure (perhaps in terms of taking so long to execute that no other alternatives or fail-safe procedures are still feasible).

## References

- [1] Anderson, T. and Lee, P.A. "Fault Tolerance Terminology Proposals," *FTCS-12*, Santa Monica, Calif., 1982, pp. 29-33.
- [2] Azema, P., and Diaz, M. "Checking Experiments for Concurrent Systems," *FTCS-7*, June 1977, p. 206.
- [3] Azema, P., Diaz, M., and Doucet, J. "Multilevel Description Using Petri Nets," *1975 IEEE International Symposium on Computer Hardware Description Languages and Their Applications*, September 1975, pp. 188-190.
- [4] Azema, P., Valette, R. and Diaz, M. "Petri Nets as a Common Tool for Design Verification and Hardware Simulation," *13th IEEE Design Automation Conference*, June 1976, pp. 109-116.

- [5] Berthomieu, B. and Menasche, M. "An Enumerative Approach for Analyzing Time Petri Nets," *Proc. of 1983 IFIP Congress*, Paris, September 1983.
- [6] Coolahan, J.E. and Roussopoulos, N. "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Trans. on Software Engineering*, vol. SE-9, no. 5, Sept. 1983, pp. 603-616.
- [7] Estrin, G. "A Methodology for Design of Digital Systems -- Supported by SARA at the Age of One", *AFIPS, Proceedings of the National Computer Conference*, 1978.
- [8] Hack, M. "Analysis of Production Schemata by Petri Nets," Technical Report 94, Project MAC, MIT, February 1972.
- [9] Hammer, W. *Handbook of System and Product Safety*, Prentice Hall, 1972.
- [10] Kopetz, H. "The Failure Fault (FF) Model," *FTCS-12*, Santa Monica, Calif., June 1982, pp. 14-17.
- [11] Laprie, J.C. and Costes, A. "Dependability: A Unifying Concept for Reliable Computing," *FTCS-12*, Santa Monica, Calif., June 1982, pp. 18-21.
- [12] Leveson, N.G. and Harvey, P.R. "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 5, Sept. 1983.
- [13] Leveson, N.G. "Software Safety in Process-Control Systems," *IEEE Computer*, February 1984.
- [14] Merlin, P.M. "A Study of the Recoverability of Computing Systems," Ph.D. Thesis, Information and Computer Science Department, University of California, Irvine, 1974.
- [15] Merlin, P.M. and Farber, D.J. "Recoverability of Communication Protocols -- Implications of a Theoretical Study," *IEEE Transactions on Communications*, vol. COM-24, no. 9, September 1976, pp. 1036-1043.
- [16] MIL-STD-882a, System Safety Program Requirements, U.S. Department of Defense, April 1984.
- [17] Nelson, R.A., Haitb, L.M., and Sheridan, P.B. "Casting Petri Nets into Programs," *IEEE Trans. on Software Engineering*, vol. SE-9, no. 5, September, 1983, pp. 590-602.
- [18] Noe, J.D. "Hierarchical Modeling with Pro-Nets," *National Electronics Conference*, vol. 23, October 1978, pp. 155-160.
- [19] Peterson, J.L. *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

- [20] Ramamoorthy, C.V. and Ho, G.S. "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets," *IEEE Trans. on Software Engineering*, vol. SE-6, September 1980, pp. 440-449.
- [21] Ramchandani, C. "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets," Ph.D. Thesis, Project MAC Report No. MAC-TR-120, MIT, 1974.
- [22] Razouk, R.R. "Computer-Aided Design and Evaluation of Digital Computer Systems," Ph.D. Thesis, University of California, Los Angeles, February 1981.
- [23] Razouk, R.R. "The Derivation of Performance Expressions for Communication Protocols from Timed Petri Net Models," Technical Report #211, Information and Computer Science Department, University of California, Irvine, November 1983.
- [24] Rose, C.W. and Albarran, M. "Modeling and Design Description of Hierarchical Hardware/Software Systems," *Proc. 12th Design Automation Conference*, Boston, 1975.
- [25] Rose, M. "Modeling and Analysis of Concurrent Systems using Contour/Transistion-Nets, Ph.D. Dissertation, Information and Computer Science Department, University of California, Irvine, 1984.
- [26] Sifakis, J. "Petri Nets for Performance Evaluation," *Measuring, Modeling, and Evaluating Computer Systems*, Proceedings of the 3rd Symposium, IFIP Working Group 7.3, H. Beilner and E. Gelenbe (eds.), North Holland, 1977, pp. 75-93.
- [27] Vesely, W.E., Goldberg, F.F., Roberts, N.H., and Haasl, D.F. *Fault Tree Handbook*, NUREG-0492, U.S. Nuclear Regulatory Commission, January 1981.
- [28] Zuberek, W.M. "Timed Petri Nets and Preliminary Performance Evaluation," *Proc. IEEE 7th Annual Symposium on Computer Architecture*, La Baule, France, 1980, pp. 88-96.